Introduction to Real-Time Maude

Peter C. Ölveczky

University of Illinois at Urbana-Champaign

and

University of Oslo

Based on joint work with José Meseguer

Overview

- Real-Time Maude extends Maude 2.1 to specify and analyze real-time and hybrid systems
- Specification formalism based on the real-time rewrite theory model
- Emphasizes expressiveness and ease of specification, and performance
- Supports specification of object-based real-time systems
- Extends Maude's formal analysis capabilities to real-time systems:
  - symbolic simulation for prototyping
  - exhaustive state space exploration
  - temporal logic model checking
- Implemented in Maude as an extension of Full Maude 2.1

Motivation

- Specify and analyze large real-time systems
- Real-Time Maude complements
  - finite-control-automaton-based tools (HyTech, UPPAAL, Kronos)
    - expressiveness, ease of specification, data types, OO
  - verification tools (e.g. STeP)
    - different validation methods, OO
  - informal techniques
  - networks simulation tools and testbeds
    - more abstract
    - wider range of analysis techniques

Comparison with Automaton-based Tools

- UPPAAL, HyTech, and Kronos successful model checking tools for real-time systems
  - specification formalism: finite control linear hybrid automata
    - restricted formalism ensures a way of treating dense time
  - specification formalism often too restrictive for larger systems
    - communication; "infinite-control" systems; data types and OO
- Real-Time Maude: ease and generality of specification
  - expressiveness: system properties not decidable in general
  - can not deal with dense time by "clock region"-like constructions
  - choice of time sampling strategies to handle dense time
    - may not cover all "interesting" behaviors
    - . . . but often sufficient in practice

Real-Time Maude Specification

- Algebraic equational specification defines the state space
  - must include specification of (dense or discrete) time domain
- (Ordinary) rewrite rules model instantaneous change
- Tick rewrite rules

  $\{t\}$ `=>` $\{t'\}$ `in time` t'' `if` $cond$

  model time elapse
- Global state of the form $\{u\}$ to ensure uniform time elapse
- Many well known models of real-time and hybrid systems can be naturally expressed in this way

Details

- State t must have sort `System`
- $\{t\}$ then has sort `GlobalSystem`
- Time domain must have sort `Time`, and must interpret `zero`, `_plus_`, and `_monus_` (!), etc. on the time domain
- Useful predefined modules for time domains can be imported:
  - `NAT-TIME-DOMAIN`, natural numbers
  - `POSRAT-TIME-DOMAIN`, nonnegative rationals for dense time
  - `NAT-TIME-DOMAIN-WITH-INF`, adds a constant `INF` of supersort `TimeInf`
  - `POSRAT-TIME-DOMAIN-WITH-INF`

Example: One Clock

(Retrograde) clock which shows the time:

- global state $\{\texttt{clock}(r)\}$
- dense time domain
- clock can stop at any time due to battery failure
- $\texttt{clock}(24)$ should be reset to $\texttt{clock}(0)$

Example: A Clock

```
(tmod DENSE-CLOCK is pr POSRAT-TIME-DOMAIN .
  ops clock stopped-clock : Time -> System .
  vars R R' : Time .
  crl [tickWhenRunning] :
      {clock(R)} => {clock(R + R')} in time R'
         if  R' <= 24 - R  [nonexec] .
  rl [tickWhenStopped] :
     {stopped-clock(R)} => {stopped-clock(R)}
         in time R'  [nonexec] .
  rl [reset] :   clock(24) => clock(0) .
  rl [batteryDies] :
     clock(R) => stopped-clock(R) .
endtm)
```

Challenge: Dense Time

- Tick rules “cover” the dense time domain
- . . . but are not executable
- Specification formalism too general for “clock region” constructions
- Tick rules often of form

```
{t}  =>  {t'} in time x if .../\ x <= u /\ ..
```

  or similar, called admissible, for x a new variable

- Choice of time sampling strategies to treat admissible tick rules, including
  - advance time by time Δ
  - advance time as much as possible

Time Sampling Strategies

- Real-Time Maude execution of admissible tick rules relative to selected time sampling strategy
- May not cover all “interesting” behaviors
- Time sampling often sufficient in practice
  - usually discrete time domain in communication, scheduling, etc.
  - things happen because timers expire, messages are read, etc.

Example: Setting a Time Sampling Strategy

- Set time to advance by 1 in each tick rule application:

```
Maude> (set tick def 1 .)
```

  - all subsequent Real-Time Maude analysis performed w.r.t. this strategy

- Alternative time sampling strategy advances time as much as possible:

```
Maude> (set tick max .)
```

Formal Analysis I: Symbolic Simulation

Timed rewriting simulates one possible behavior:

```
Maude> (trew {clock(0)} in time <= 100 .)

Result ClockedSystem :
  {stopped-clock(24)} in time 100
```

- can trace the rewrite steps

Formal Analysis II: Search and Model Checking

- To analyze all behaviors, relative to the chosen time sampling strategy, from initial state
  - search for reachable states matching a given pattern
  - linear temporal logic model checking
- Reachable state space often infinite in larger systems:
  - search/model check up to a given duration
  - reachable state space then becomes finite (most often)
- When reachable state space finite: unbounded search/model checking
- No guarantee that all “interesting” behaviors covered
  - false positives possible
  - a counterexample is a valid counterexample

Example: Search

- Can `{clock(`r`)}` (with $r > 24$) be reached from `{clock(0)}` within time `100`?

```
(tsearch [1] {clock(0)} =>* {clock(R:Time)}
                 such that R:Time > 24
                     in time <= 100 .)
```

- Finite reachable state space: unbounded search:

```
(utsearch [1] {clock(0)} =>* {clock(25)} .)
```

- State `{clock(1/2)}` not found because of time sampling:

```
(utsearch [1] {clock(0)} =>* {clock(1/2)} .)
```

Temporal Logic Model Checking

- Propositional linear temporal logic
- Uses Maude's high-performance model checker
- Not metric temporal logic
  - propositions may involve elapsed time ("clocked properties")
- Time bound $\Rightarrow$ termination of model checking
- Provides counterexamples
- Propositions and formulas are defined in a module which imports `TIMED-MODEL-CHECKER` and the module to analyze

Example: Defining Propositions

- `clock-dead`: holds if clock has stopped

```
op clock-dead : -> Prop [ctor] .
vars R R' : Time .
eq {stopped-clock(R)} |= clock-dead = true .
```

- `clock-is(`r`)`: running clock shows r

```
op clock-is : Time -> Prop [ctor] .
eq {clock(R)} |= clock-is(R') = (R == R') .
```

- `clockEqualsTime`: running clock shows the current time elapse

```
op clockEqualsTime : -> Prop [ctor] .
eq {clock(R)} in time R' |= clockEqualsTime
      = (R == R') .
```

Example: Temporal Logic Model Checking

- The clock is never 25:

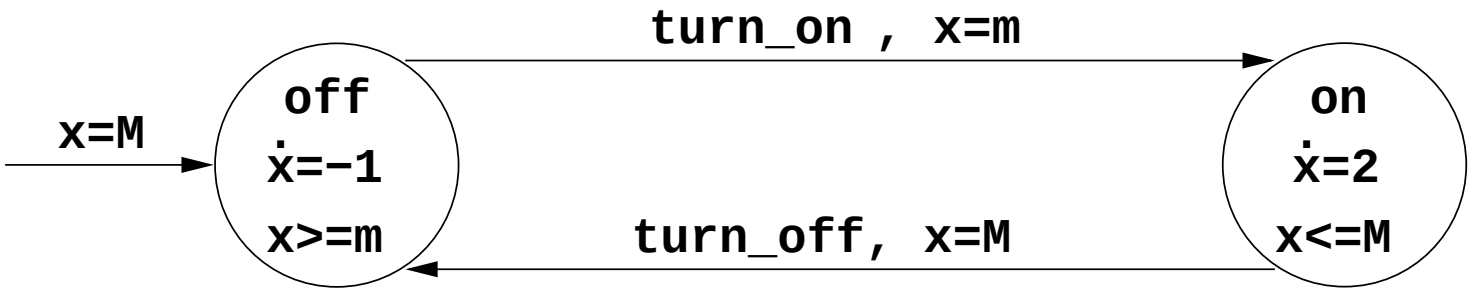
```
(mc {clock(0)} |=u [] ~ clock-is(25) .)
```

- Clock shows elapsed time until time 24 or until it dies:

```
(mc {clock(0)} |=t clockEqualsTime U
                    (clock-is(24) \/ clock-dead)
          in time <= 100 .)
```

Example: A Thermostat

A very simple thermostat example where a heater is either turned on or off:



Temperature increases by 2 degrees/time unit when heater turned on, and decreases by 1 degree/time unit when heater turned off

Exercise I: A Thermostat

1. Start Real-Time Maude
2. Model such a simple thermostat, where the thermostat must maintain a temperature should be between 62 and 74 degrees (it's Fahrenheit)
3. Global state could have form $location, temp$
   - remember: $loc, temp$ must then have sort `System`
4. Use a dense time domain
5. Define a time sampling strategy which increases time by 1/2 in each tick step
6. Define a suitable initial state
7. Execute the specification with timed rewriting (trew) to simulate one behavior from the initial state within time 1000

8. Can you use untimed search on this specification with this time sampling strategy?

9. Use search to show that it is not possible to reach a state where the temperature is too cold or too warm

10. Is it possible to reach a state where the temperature is 71 degrees?

11. (If familiar with temporal logic:) import TIMED-MODEL-CHECKER and the thermostat, and define a proposition `goodTemp` which holds for all states where the temperature is within desired interval

12. Use (untimed or timed?) temporal logic model checking to prove that the temperature is always OK

13. Set the time sampling strategy to a “maximal” strategy, and repeat the above rewriting, search, and model checking

Object-Based Specification

Real-Time Maude particularly well suited for specifying object-based real-time systems

- explicit support for object-based specification in object-oriented timed modules (syntax `(tomod ... endtom)`)
- all larger Real-Time Maude case studies object-based
- developed specification techniques for OO systems

OO Specification Techniques

Specification techniques useful in all OO case studies:

- tick rule(s) usually similar to

```
crl [tick] :
    {C:Configuration}  =>
       {delta(C:Configuration, R)} in time R
    if R <= mte(C:Configuration) [nonexec] .
```

  - R new variable $\Rightarrow$ time domain covered
  - `delta` models effect of time elapse
  - `mte` gives maximum possible time elapse before some instantaneous action must be taken
  - `delta` and `mte` are frozen operators
  - `delta` and `mte` distribute over the elements in a configuration

Example: Many OO Clocks

Object-oriented specification of multiple clocks:

```
(tomod MANY-DENSE-OO-CLOCKS is
  protecting POSRAT-TIME-DOMAIN-WITH-INF .

class Clock | state : ClockState,
              value : Time .
sort ClockState .
ops running stopped : -> ClockState [ctor] .
var O : Oid .
rl [reset] :
   < O : Clock | state : running, value : 24 >
 =>  < O : Clock | state : running, value : 0 > .
```

Example: Many OO Clocks II

```
rl [failure] :
   < O : Clock | state : running >  =>
      < O : Clock | state : stopped > .

vars C C' : NEConfiguration .
vars R R' : Time .

crl [tick] :
    {C}  =>  {delta(C, R)} in time R
       if R le mte(C) [nonexec] .
```

Example: Many OO Clocks III

```
op mte : NEConfiguration -> TimeInf [frozen (1)] .
eq mte(C C') = min(mte(C), mte(C')) .
eq mte(< O : Clock | state : running,
                     value : R >) = EXERCISE!! .
eq mte(< O : Clock | state : stopped,
                     value : R >) = EXERCISE!! .
```

Example: Many OO Clocks IV

```
op delta : NEConfiguration Time ->
           NEConfiguration [frozen (1)] .
eq delta(C C', R) = delta(C, R)  delta(C', R) .
eq delta(< O : Clock | state : running,
                       value : R >, R') =
                        EXERCISE!! .
eq delta(< O : Clock | state : stopped >, R) =
                        EXERCISE!! .
endtom)
```

Exercise II: OO Clocks

1. Complete the above specification of OO clocks by defining `mte` and `delta` on `Clock` objects
2. Define an initial state with three clocks

```
op initState : -> GlobalSystem .
eq initState = { ... } .
```

   - remember to define some values of sort `Oid` for the object names

3. Define a suitable time sampling strategy
4. Execute your specification using timed rewriting
5. Search for a state where some clock has `value > 24`

Rest of Exercises

Simple ways of “computing” the round trip times between pairs of objects

- basis for many communication protocols
- OO: each “node” in the network corresponds to an object
- need a “clock” component in each node (no death or reset)
- need to model time it takes a message to travel between a pair of objects
- look at different settings
- “real” Real-Time Maude case study uses sophisticated modeling of delays and congestion in links

Exercise III: Clock Attribute

1. Specifiy a OO system where each `Node` object has one attribute `clock` which shows the accumulated time (no reset/death of clocks)
   - can use discrete time domain
2. Define a time sampling strategy which increases time by 1
3. Define an initial state with three `Node` objects, each with `clock` value 0
4. Execute the specification
5. Use `search` to show that a state where one clock has value 100 can be reached
6. Use search to show that all the nodes have the same clock values all the time (possibly up to time 100)

Modeling Message Delays

To model message "delays" can have an operator

```
op dly : Msg Time -> Msg .
```

such that `dly(`m`,` r`)` denotes a "message" which will become m in time r from now

- Nice idea:

```
op dly : Msg Time -> Msg [idr: 0] .
```

  - `dly(`m`,0)` is now seen exactly as m by Real-Time Maude!

Exercise IV: Delays

Define the function `delta` on `dly`-messages and on normal messages

- `mte` is more tricky
- depends on the context

First RTT Protocol

Class `Node`:

```
class Node | clock : Time, nbr : Oid,
             dlyToNbr : Time, rttToNbr : Time .
```

- `nbr` is the address to the neighbor node
- `dlyToNbr` is the exact time it takes for a message to reach the neighbor
- `rtt` is the desired RTT value to the neighbor we want to find

First RTT Protocol (cont.)

Protocol:

1. A `findRtt(`n`)` message kicks off the protocol for node n
2. When node n reads the `findRtt(`n`)` message, it sends an `rttReq(`r`)` message to its neighbor, where r is the "timestamp," its current `clock` value
3. When a node n' reads an `rttReq(`r`)` message, it responds with an `rttAck(`r`)` (with same r) message back to its neighbor, with appropriate delay
4. When a node receives a `rttAck(`r`)` message, it computes the desired `rttToNbr` value as the difference between the timestamp r and its current `clock` value

First RTT Protocol (cont.)

A typical initial state could be

```
ops n1 n2 n3 n4 : -> Oid .
op initState : -> GlobalSystem .
eq initState =
   {findRtt(n1)    findRtt(n3)    findRtt(n4)
     < n1 : Node | clock : 0, nbr : n3,
                   dlyToNbr : 2, rttToNbr : 0 >
     < n2 : Node | clock : 0, nbr : n4,
                   dlyToNbr : 1, rttToNbr : 0 >
     < n3 : Node | clock : 0, nbr : n1,
                   dlyToNbr : 3, rttToNbr : 0 >
     < n4 : Node | clock : 0, nbr : n2,
                   dlyToNbr : 2, rttToNbr : 0 >} .
```

Exercise V: Simple RTT

1. What should be the “result” of the system from `initState`?
2. Specify and execute the protocol with the given initial state
3. Check whether each state reachable in exactly time 6 has the correct `rttToNbr` values
   - hint: search for a state reachable in time 6 with some incorrect `rttToNbr` value

Example: New RTT Setting

A message may now take any amount of time $\geq$ the minimum delay to reach its destination

- nondeterministic traveling time for a message (models different network load, etc.)
- hint: consider `mte` on messages
- the actual protocol is the same!

Exercise VI: New RTT Setting

1. Modify your previous solution to the new setting, and execute your specification
2. Show that a message can indeed travel any time by
   - proving that it is possible to reach a state in which both nodes have recorded an `rttToNbr` value 2
   - proving that it is possible to reach a state where `n1` has recorded `rttToNbr` 4 and `n2` has recorded `rttToNbr` value 5

   from initial state

```
{findRtt(n1)    findRtt(n2)
 < n1 : Node | clock : 0, nbr : n2,
               dlyToNbr : 1, rttToNbr : 0 >
 < n2 : Node | clock : 0, nbr : n1,
               dlyToNbr : 1, rttToNbr : 0 >}
```

Example: New RTT Protocol

- Communication model unchanged
- Protocol has changed:
  - if an `rttAck` message is not received within time `MAX-RTT`, then something went very badly, and we instead initiate a new round of the protocol to find a OK rtt value
  - each time a node sends a `rttReq` message, it also turns on a timer which must “ring” in time `MAX-RTT`
  - if an `rttAck` message is received which indicates an rtt value ¡ `MAX-RTT`: everything OK, record rtt value and turn off timer
  - if timer rings: start new attempt and set timer to ring in time `MAX-RTT`
  - “old” messages are just ignored

Exercises VII: New RTT Protocol

(If time)

1. Model the protocol described above in Real-Time Maude
   - hint: a timer can be modeled by an attribute with a time value r if it must "ring" in time r, and with a value `INF` if it is turned off
2. Execute the protocol
3. Use search to show that the protocol does not record any rtt value $\geq$ `MAX-RTT` (within reasonable time)

Larger Case Studies in Real-Time Maude

- AER/NCA: new state-of-the-art protocol suite for scalable, adaptive, and reliable multicast in active networks
  - object-based formal specification from UML-like informal specification
  - Real-Time Maude analysis uncovered all design errors discovered by traditional testing, and some more
- Proposed new IETF multicast protocol
- New and advanced scheduling algorithms
- Wireless sensor network protocols

Concluding Remarks

- Formalism emphasizes generality and ease of specification
  - suitable for object-based specification
- Range of high-performance analysis capabilities
- Complements automaton-based model checkers, verification tools, informal specification techniques, and simulation tools
- Tested in substantial case studies
- Suggested OO specification techniques useful in practice
- Tool, user manual, related papers, and examples available from

  `http://www.ifi.uio.no/RealTimeMaude`

  and from the Maude page

Future Work

- Identify cases in which a time sampling strategy covers all behaviors
- Develop other abstraction techniques
- Deal with dense time
- Stochastic time increase strategies for simulation purposes
- Case studies
- . . . and much more